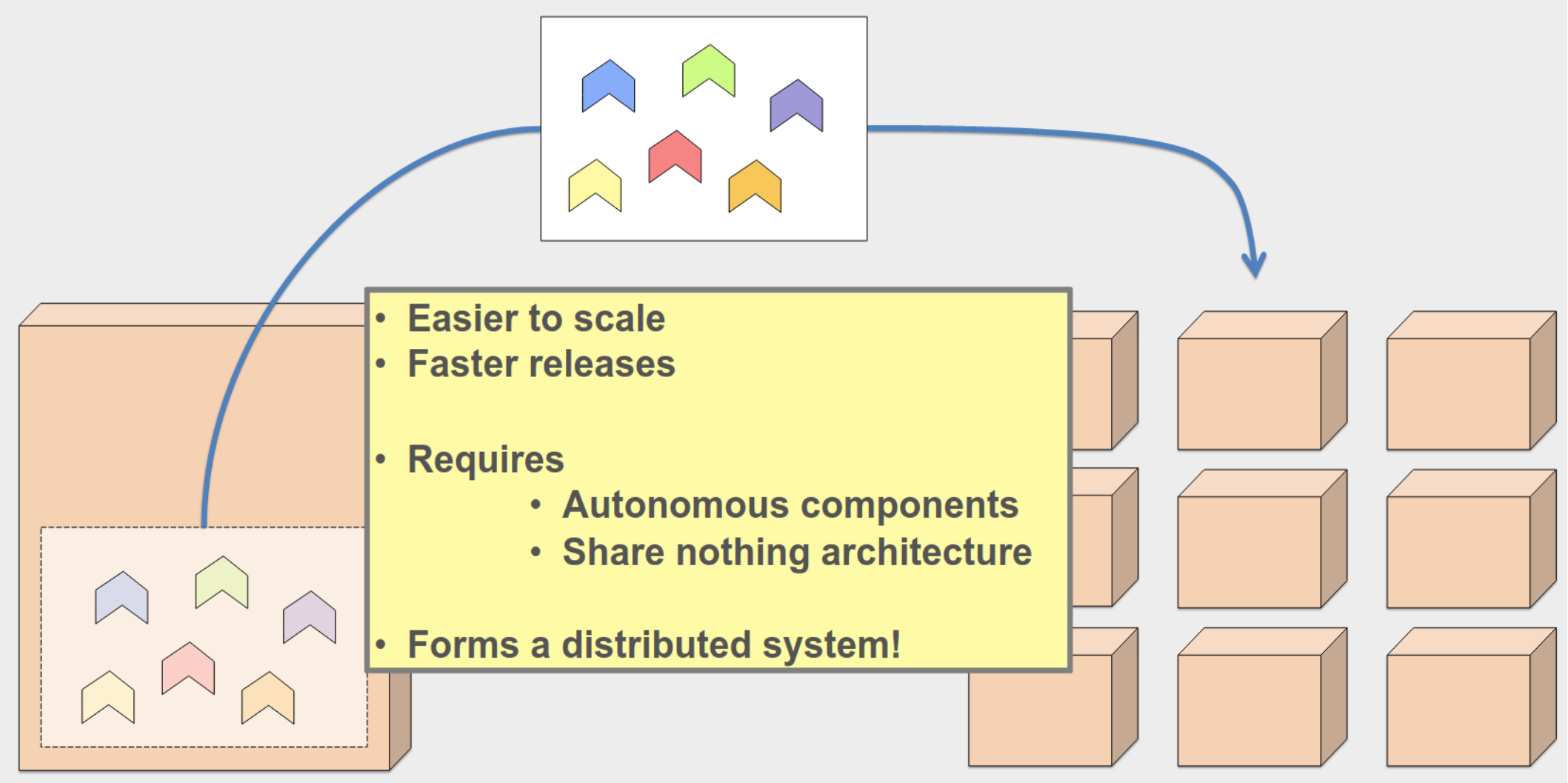
Microservices Architecture for Java

Introduction

- Why
- What is a microservice-based architecture?
- Challenges with microservices
- Design patterns for handling challenges
- Software enablers that can help us handle these challenges

Introduction:Why

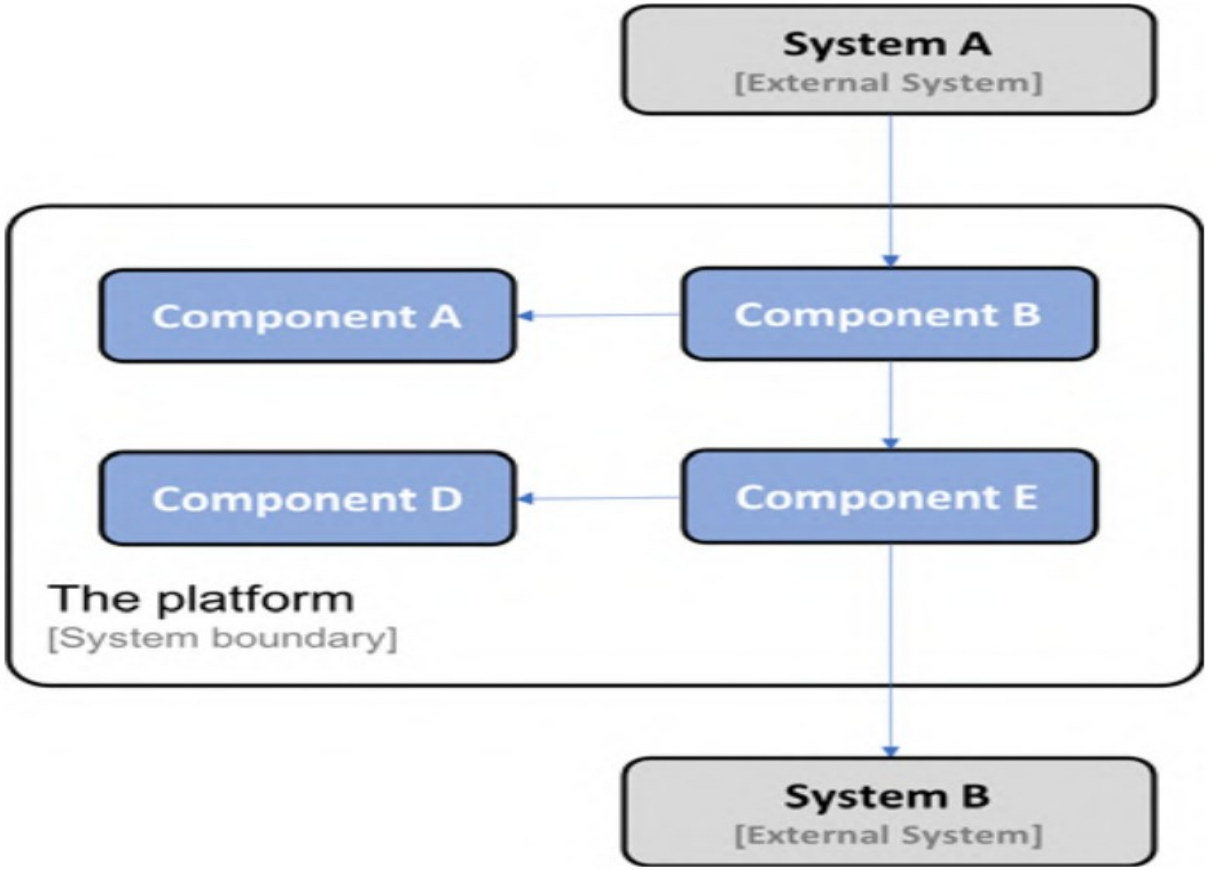


Introduction:Why

- **Benefits of autonomous software components**
  - A customer can deploy parts of the platform in its own system landscape, integrating it with its existing systems using its well-defined APIs.
  - The following is an example where one customer decided to deploy Component A, Component B, Component D, and Component E from the platform and integrate them with two existing systems in the customer's system landscape, System A and System B

Introduction:Why

- **Benefits of autonomous software components**

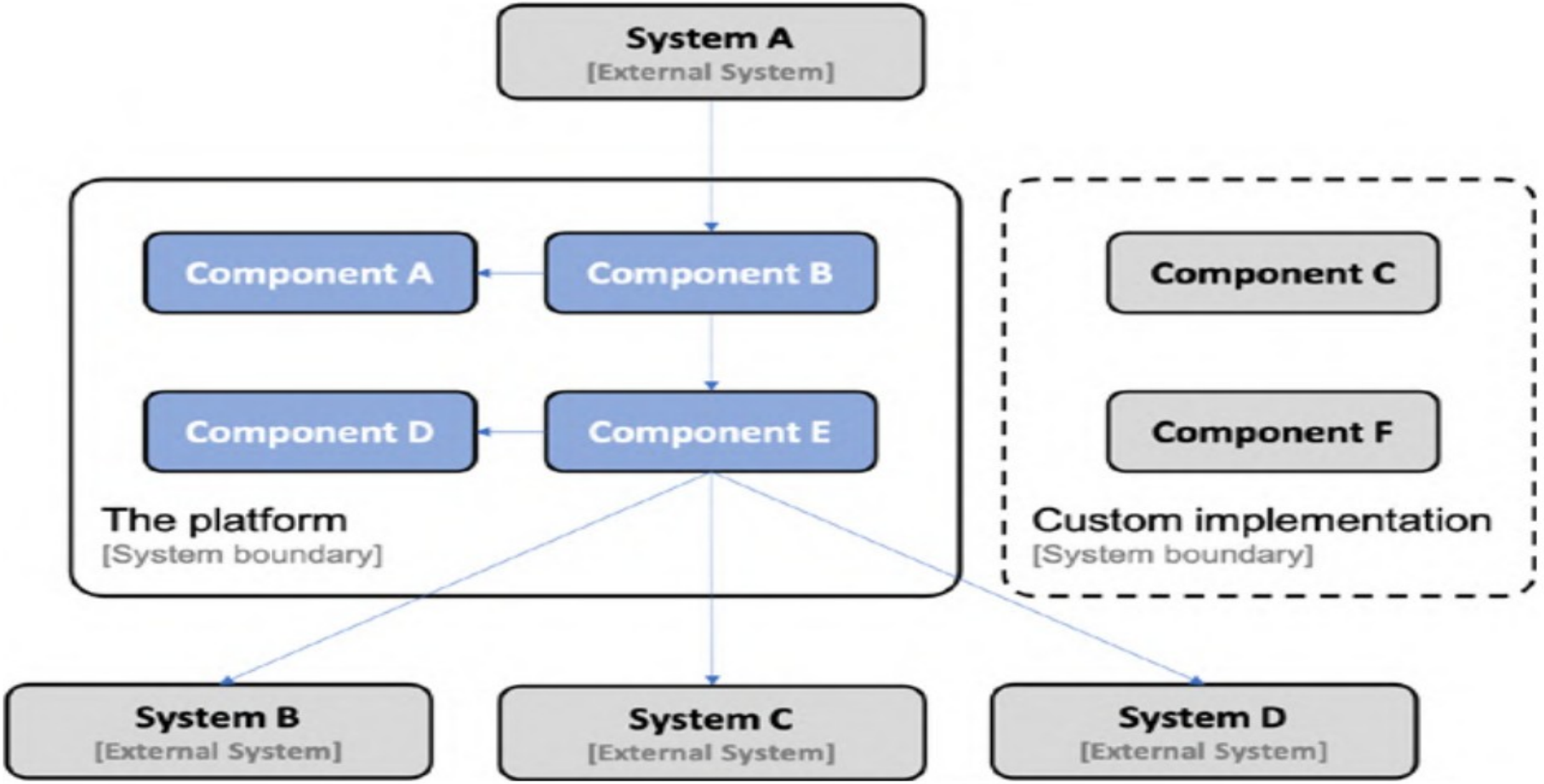


Introduction:Why

- **Benefits of autonomous software components**
  - Another customer can choose to replace parts of the platform's functionality with implementations that already exist in the customer's system landscape, potentially requiring some adoption of the existing functionality in the platform's APIs.
  - The following is an example where a customer has replaced Component C and Component F in the platform with their own implementation:

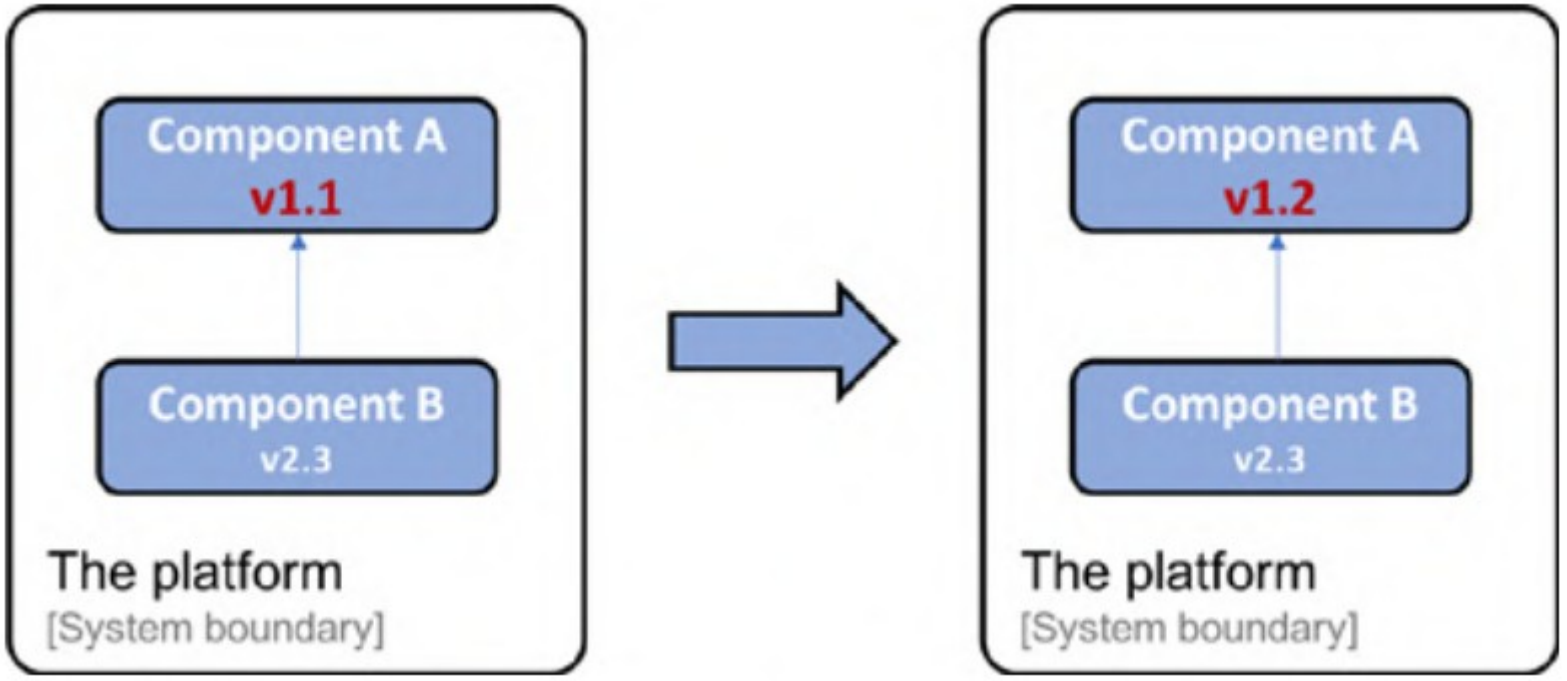
Introduction:Why

- **Benefits of autonomous software components**



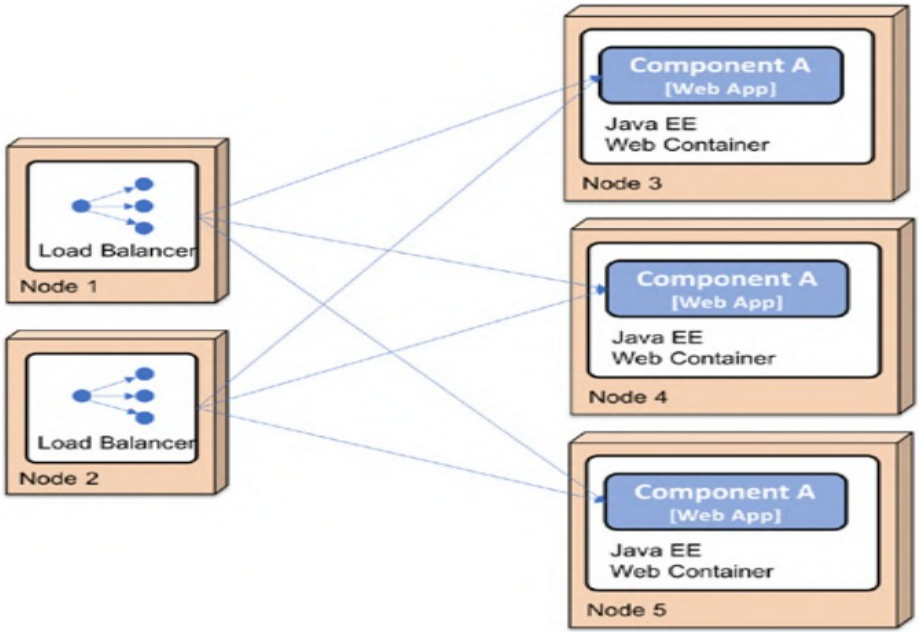
Introduction:Why

- **Benefits of autonomous software components**
  - Each component in the platform can be delivered and upgraded separately. Thanks to the use of well-defined APIs, one component can be upgraded to a new version without being



Introduction:Why

- **Benefits of autonomous software components**
  - Thanks to the use of well-defined APIs, each component in the platform can also be scaled out to multiple servers independently of the other components. Scaling can be done either to meet high availability requirements or to handle higher volumes of requests.



Introduction:Challenges

- **Challenges**
  - Decomposing the platform introduced a number of new challenges that we were not exposed to (at least not to the same degree) when developing more traditional, monolithic applications:
    - **Scaling**:Time consuming and error prone
    - **Chain of failures**
    - **Configuration** in all the instances of the components consistent and up to date quickly became a problem
    - **Monitoring**
    - **Logs**

Introduction:Challenges

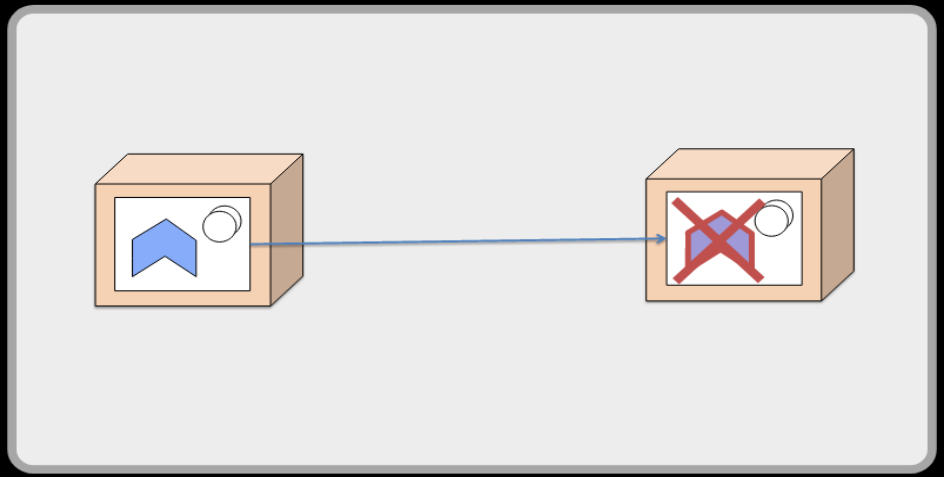
CHALLENGES

DISCOVERY SERVER
WHERE ARE THE SERVICES?
WHICH SERVICE TO CALL?

EDGE SERVER
HOW TO HIDE PRIVATE SERVICES?
HOW TO PROTECT PUBLIC SERVICES?

RESILIENCE
HOW TO HANDLE FAULTS?
- SLOW OR NO RESPONSE
- TEMPORARY FAULTS
- OVERLOAD

DISTRIBUTED TRACING
WHO IS CALLING WHO?



CENTRALIZED CONFIGURATION
WHERE IS MY CONFIGURATION?
ARE ALL SERVICES
CONFIGURATION UP TO DATE?

TRAFFIC MANAGMENT
HOW TO CONTROL ROUTING?
- RATE LIMITING
- CANARY & BLUE/GREEN UPGRADES

SERVICE MANAGEMENT
HOW TO
- DEPLOY SERVICES?
- SCALE SERVICES?
- UPGRADE SERVICES?
- RESTART FAILING SERVICES?

OBSERVABILITY
HOW ARE MY SERVICES PERFORMING?

LOG ANALYSIS
WHERE ARE THE LOGS?
HOW TO CORRELATE LOGS
FROM DIFFERENT SERVICES?

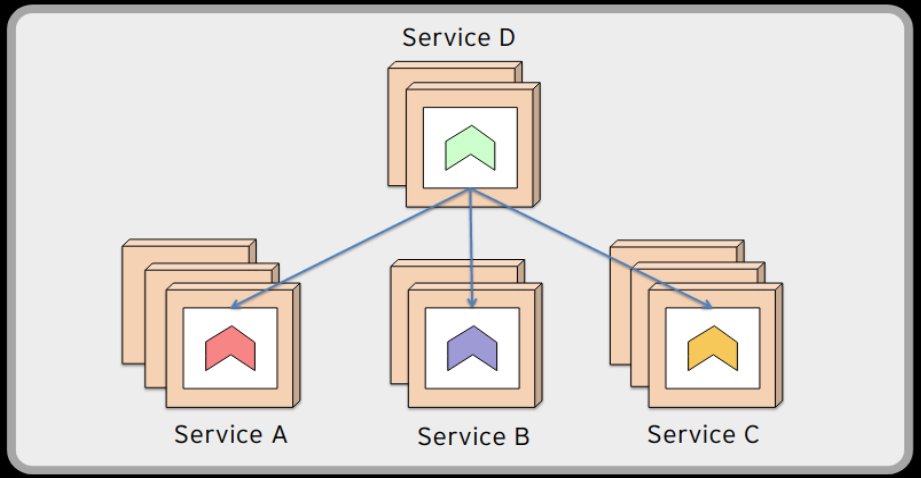
MONITORING
WHAT HARDWARE RESOURCES ARE USED?

Introduction:Challenges

REQUIRED CAPABILITIES!

DISCOVERY SERVER
WHERE ARE THE SERVICES?
WHICH SERVICE TO CALL?

EDGE SERVER
HOW TO HIDE PRIVATE SERVICES?
HOW TO PROTECT PUBLIC SERVICES?



RESILIENCE
HOW TO HANDLE FAULTS?
- SLOW OR NO RESPONSE
- TEMPORARY FAULTS
- OVERLOAD

DISTRIBUTED TRACING
WHO IS CALLING WHO?

CENTRALIZED CONFIGURATION
WHERE IS MY CONFIGURATION?
ARE ALL SERVICES
CONFIGURATION UP TO DATE?

TRAFFIC MANAGMENT
HOW TO CONTROL ROUTING?
- RATE LIMITING
- CANARY & BLUE/GREEN UPGRADES

SERVICE MANAGEMENT
HOW TO
- DEPLOY SERVICES?
- SCALE SERVICES?
- UPGRADE SERVICES?
- RESTART FAILING SERVICES?

OBSERVABILITY
HOW ARE MY SERVICES PERFORMING?

LOG ANALYSIS
WHERE ARE THE LOGS?
HOW TO CORRELATE LOGS
FROM DIFFERENT SERVICES?

MONITORING
WHAT HARDWARE RESOURCES ARE USED?

Introduction:Microservices

- **Defining a microservice**
  - **Faster development**, enabling continuous **deployments**
  - Easier to **scale**, manually or automatically
  - It must conform to a **shared-nothing architecture**; that is, microservices don't share data in databases with each other!
  - It must only communicate through **well-defined interfaces**, either using APIs and synchronous services or preferably by sending messages asynchronously. The APIs and message formats used must be stable, well- documented, and evolve by following a defined versioning strategy.
  - It must be deployed as **separate runtime processes**. Each instance of a microservice runs in a separate runtime process, for example, a Docker container.
  - Microservice instances are **stateless** so that incoming requests to a microservice can be handled by any of its instances.

Introduction:Microservices

- **Defining a microservice**
  - **How big should a microservice be?**
  - Small enough to fit in the head of a developer
  - Big enough to not jeopardize performance (that is, latency) and/or data consistency (SQL foreign keys between data that's stored in different microservices are no longer something you can take for granted)
  - to summarize, a microservice architecture is, in essence, an architectural style where we decompose a monolithic application into a group of cooperating autonomous software components.
    - The motivation is to enable faster development and to make it easier to scale the application.

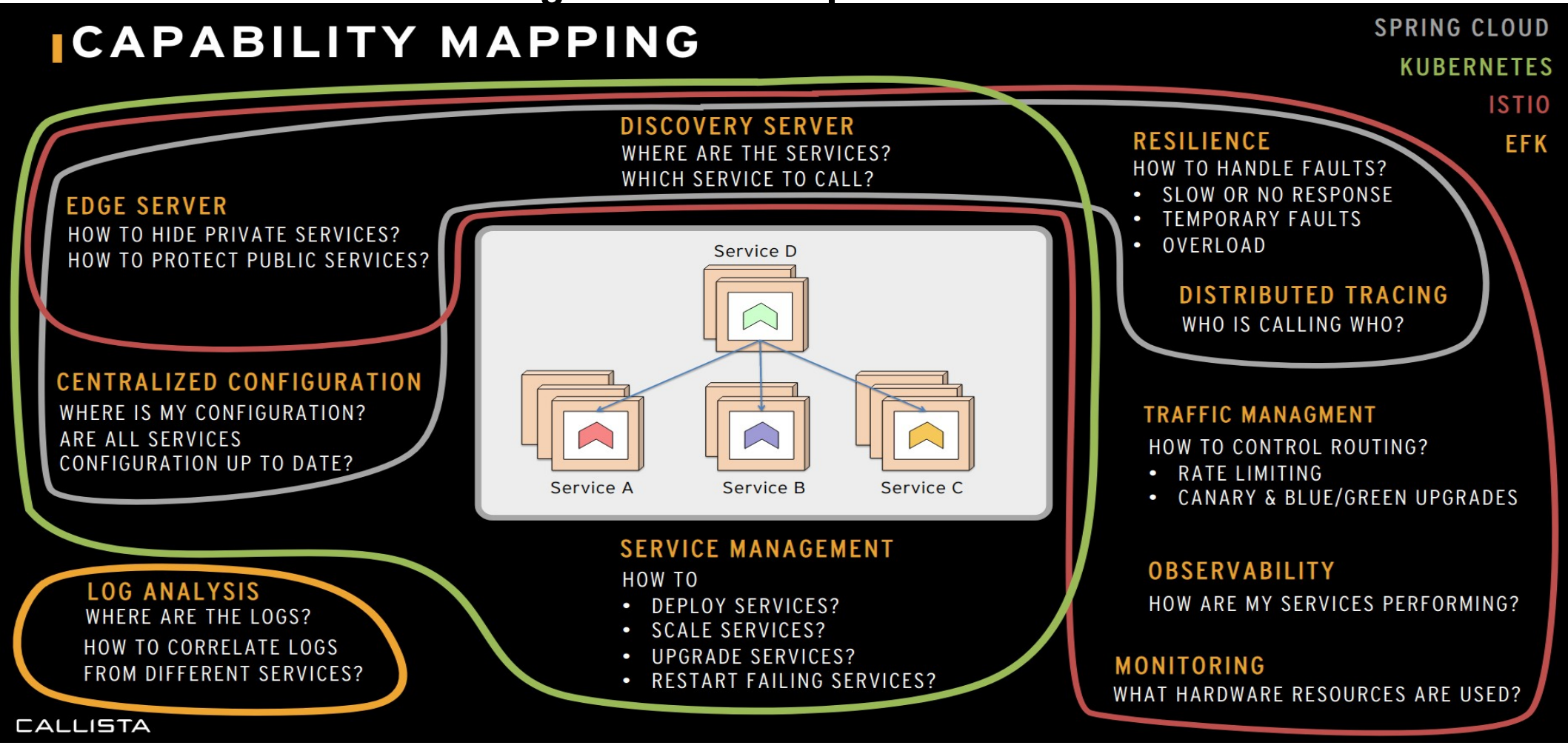
Introduction:Microservices Design patterns

- **Microservices Design Patterns:Open source to the rescue**

Design Pattern	Spring Boot	Spring Cloud	Kubernetes	Istio
Service discovery		Netflix Eureka and Spring Cloud LoadBalancer	Kubernetes `kube-proxy` and service resources	
Edge server		Spring Cloud and Spring Security OAuth	Kubernetes Ingress controller	Istio ingress gateway
Reactive microservices	Project Reactor and Spring WebFlux			
Central configuration		Spring Config Server	Kubernetes `ConfigMaps` and Secrets	
Centralized log analysis			Elasticsearch, Fluentd, and Kibana Note: Actually not part of Kubernetes, but can easily be deployed and configured together with Kubernetes	

Introduction:Microservices Design patterns

- **Microservices Design Patterns:Open source to the rescue**



Introduction:Microservices:Important Considerations

- **Important Consideration**
  - Importance of DevOps
  - Organizational aspects and Conway's law
  - Decomposing a monolithic application into microservices
  - Decomposing a monolithic application into microservices: One of the most difficult decisions (and expensive if done wrong) is how to decompose a monolithic application into a set of cooperating microservices. If this is done in the wrong way, you will end up with problems such as the following
    - Slow delivery
    - Bad performance
    - Inconsistent data

Introduction:Microservices:Important Considerations

- **Important Consideration**
  - Importance of API design
  - Migration paths from on-premises to the cloud
  - Good design principles for microservices, the 12-factor app(https://12factor.net)

Introduction:Microservices:Implementation

- **Implementation**
  - Develop microservices that contain business logic based on plain **Spring Beans** and expose **REST APIs using Spring WebFlux**. The APIs will be documented based on the **OpenAPI** specification using **springdoc-openapi**. To make the data processed by the microservices persistent, we will use **Spring Data** to store data in both SQL and NoSQL databases.

Introduction:Microservices:Implementation

- **Implementation**
  - Spring Boot
  - Spring WebFlux
  - springdoc-openapi
  - Spring Data
  - Spring Cloud Stream
  - Docker

Introduction:Microservices:SpringBoot

- **SpringBoot**
  - Spring Boot, and the Spring Framework that Spring Boot is based on, is a great framework for developing microservices in Java.
  - Spring Boot targets the fast development of production-ready Spring applications by being **strongly opinionated** about how to set up both **core modules** from the **Spring Framework** and **third-party products**, such as **libraries** that are used for logging or connecting to a database.
  - Spring Boot does that by applying a **number of conventions** by default, **minimizing the need for configuration**.
    - This pattern is known as convention over configuration

Introduction:Microservices:SpringBoot

- **SpringBoot:@SpringBootApplication**
  - The magic @SpringBootApplication annotation
    - The convention-based autoconfiguration mechanism can be initiated by annotating the application class, that is, the class that contains the static main method, with the @SpringBootApplication annotation

```
@SpringBootApplication
public class MyApplication {

  public static void main(String[] args) {
    SpringApplication.run(MyApplication.class, args);
  }
}
```

Introduction:Microservices:SpringBoot

- **SpringBoot:ComponentScan**
  - Another component in the application can get this component automatically injected, also known as auto-wiring, using the @Autowired annotation:

```
public class AnotherComponent {

  private final MyComponent myComponent;

  @Autowired
  public AnotherComponent(MyComponent myComponent) {
    this.myComponent = myComponent;
  }
```

```
@Component
public class MyComponentImpl implements MyComponent { ...
```

Introduction:Microservices:SpringBoot

- **SpringBoot:ComponentScan**
  - If we want to use components that are declared in a package outside the application's package, for example, a utility component shared by multiple Spring Boot applications, we can complement the @SpringBootApplication annotation in the application class with a @ComponentScan annotation:

```
@SpringBootApplication
@ComponentScan({"com.stillwaters.medconnect"})
public class MedconnectCompositeServiceApplication {
    1 usage
    private static final Logger LOG = LoggerFactory.getLogger(MedconnectCompositeServiceApplication.class);
    3 usages
    private final Integer threadPoolSize;
    2 usages
    private final Integer taskQueueSize;
    4 usages
    @Autowired
    MedconnectCompositeServiceIntegration integration;
```

Introduction:Microservices:SpringBoot

- **SpringBoot:Java-based configuration**
  - Override Spring Boot's default configuration or if we want to add our own configuration, we can simply annotate a class with @Configuration and it will be picked up by the component scanning mechanism we described previously

```
@Configuration
public class SubscriberApplication {

  @Bean
  public Filter logFilter() {
    CommonsRequestLoggingFilter filter = new
        CommonsRequestLoggingFilter();
    filter.setIncludeQueryString(true);
    filter.setIncludePayload(true);
    filter.setMaxPayloadLength(5120);
    return filter;
  }
```

Introduction:Microservices:SpringWebFlux

- **SpringWebFlux**
  - Spring Boot 2.0 is based on the Spring Framework 5.0, which came with built-in support for developing reactive applications.
  - The **Spring Framework uses Project Reactor** as the base implementation of its reactive support, and also comes with a new web framework, Spring WebFlux, which supports the development of **reactive, that is, non-blocking, HTTP clients and services**.

Introduction:Microservices:SpringWebFlux

- **SpringWebFlux**
  - Spring WebFlux supports two different programming models:
    - An annotation-based imperative style, similar to the already existing web framework, Spring Web MVC, but with support for reactive services
    - A new function-oriented model based on routers and handlers

Introduction:Microservices:SpringWebFlux

- **SpringWebFlux:Setup**
  - Starter dependencies

```
implementation('org.springframework.boot:spring-boot-starter-webflux')
{
 exclude group: 'org.springframework.boot', module: 'spring-boot-
 starter-reactor-netty'
}
implementation('org.springframework.boot:spring-boot-starter-tomcat')
```

  - Property files

```
server.port: 7001
```

Introduction:Microservices:SpringWebFlux

- **SpringWebFlux:Setup**
  - Rest Controller

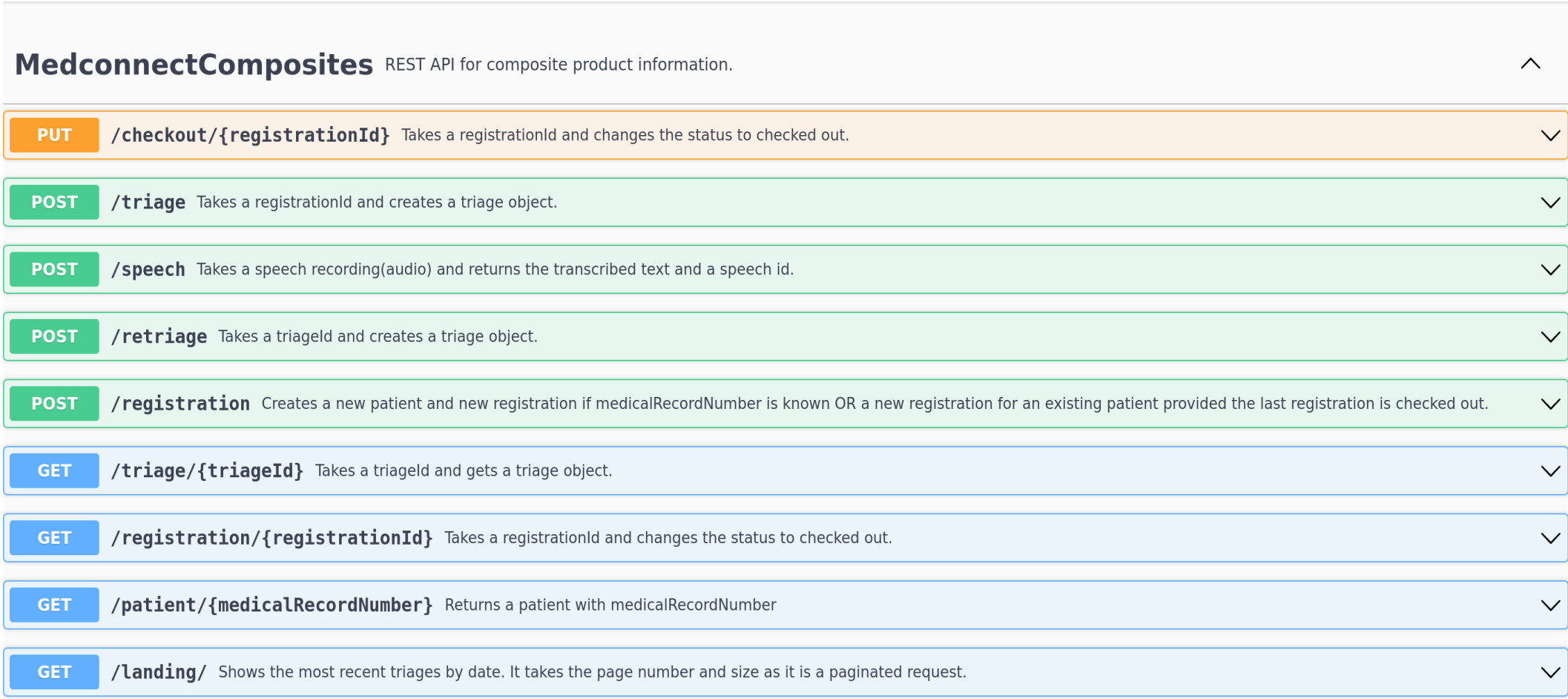
```
@RestController
public class MyRestService {

  @GetMapping(value = "/my-resource", produces = "application/json")
  List<Resource> listResources() {
    ...
  }
```

Introduction:Microservices:Springdoc-openapi

- **Springdoc-openapi**
  - springdoc-openapi is an open-source project, separate from the Spring Framework, that can create OpenAPI-based API documentation at runtime.
  - It does so by examining the application, for example, inspecting WebFlux and Swagger-based annotations.

Introduction:Microservices:Springdoc-openapi

- **Springdoc-openapi**



Introduction:Microservices:Spring Data

- **Spring Data**
  - Spring Data comes with a common programming model for persisting data in various types of database engine, ranging from traditional relational databases (**SQL databases**) to various types of **NoSQL** database engine, such as document databases (for example, **MongoDB**), key-value databases (for example, **Redis**), and graph databases (for example, **Neo4J**).

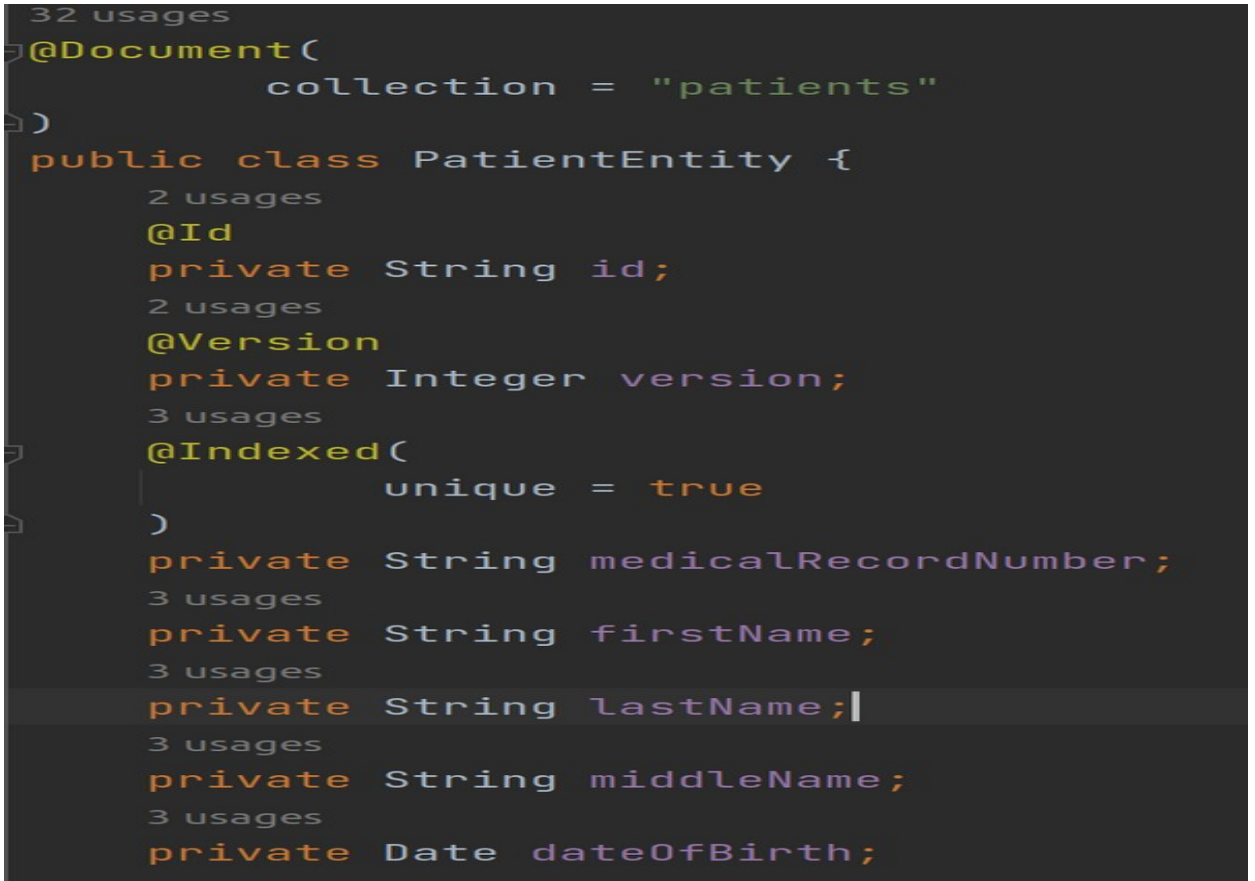
Introduction:Microservices:Spring Data

- **Spring Data**
  - An entity that will be stored in a relational database can be annotated with JPA annotations

```
@Entity
@IdClass(ReviewEntityPK.class)
@Table(name = "review")
public class ReviewEntity {
 @Id private int productId;
 @Id private int reviewId;
 private String author;
 private String subject;
 private String content;
```

Introduction:Microservices:Spring Data

- **Spring Data**
  - Spring Data MongoDB



```
32 usages
@Document(
        collection = "patients"
)
public class PatientEntity {
    2 usages
    @Id
    private String id;
    2 usages
    @Version
    private Integer version;
    3 usages
    @Indexed(
            unique = true
    )
    private String medicalRecordNumber;
    3 usages
    private String firstName;
    3 usages
    private String lastName;
    3 usages
    private String middleName;
    3 usages
    private Date dateOfBirth;
```

Introduction:Microservices:Spring Data

- **Spring Data:Repositories**
  - To specify a repository for handling the JPA entity ReviewEntity
  - ReviewEntityPK, to describe a composite primary key.

```
public interface ReviewRepository extends
  CrudRepository<ReviewEntity, ReviewEntityPK> {

  Collection<ReviewEntity> findByProductId(int productId);
}
```

```
public class ReviewEntityPK implements Serializable {
    public int productId;
    public int reviewId;
}
```

Introduction:Microservices:Spring Data

- **Spring Data:Repositories**
  - Thenaming of the method follows a naming convention defined by Spring Data that allows Spring Data to generate the implementation of this method on the fly as well.
    - If we want to use the repository, we can simply inject it and then start to use it,

```
private final ReviewRepository repository;

@Autowired
public ReviewService(ReviewRepository repository) {
 this.repository = repository;
}
```

```
public void someMethod() {
    repository.save(entity);
    repository.delete(entity);
    repository.findByProductId(productId);
```

Introduction:Microservices:Spring Data

- **SpringData: Reactive Repositories**
  - Specifying a reactive repository for handling the MongoDB entity, PatientEntity

```
public interface PatientRepository extends ReactiveCrudRepository<PatientEntity, String> {
    3 usages
    Flux<PatientEntity> findByMedicalRecordNumber(String medicalRecordNumber);

    1 usage
    @Query("{'lastName':{'$regex':'?0','$options':'i'}}")
    Flux<PatientEntity> searchByLastName(String lastName, Pageable page);

    1 usage
    @Query("{'firstName':{'$regex':'?0','$options':'i'}}")
    Flux<PatientEntity> searchByFirstName(String firstName, Pageable page);

    2 usages
    @Query("{ $or: [ { 'lastName' : {$regex:?0,$options:'i'} }, { 'firstName' : {$regex:?0,$options:'i'} } ] }")
    Flux<PatientEntity> searchByName(String name, Pageable page);
}
```

Introduction:Microservices:Spring Cloud Stream

- **Spring Cloud Stream**
  - Spring Cloud Stream provides a streaming abstraction over messaging, based on the publish and subscribe integration pattern. Spring Cloud Stream currently comes with built-in support for **Apache Kafka** and **RabbitMQ**

Introduction:Microservices:Spring Cloud Stream

- **Spring Cloud Stream**
  - The core concepts in Spring Cloud Stream are as follows:
    - **Message**: A data structure that's used to describe data sent to and received from a messaging system.
    - **Publisher**: Sends messages to the messaging system, also known as a Supplier.
    - **Subscriber**: Receives messages from the messaging system, also known as a Consumer.
    - **Destination**: Used to communicate with the messaging system. Publishers use output destinations and subscribers use input destinations. Destinations are mapped by the specific binders to queues and topics in the underlying messaging system.
    - **Binder**: A binder provides the actual integration with a specific messaging system, similar to what a JDBC driver does for a specific type of database

Introduction:Microservices:Spring Cloud Stream

- **Spring Cloud Stream:Sending**
  - To implement a publisher, we only need to implement the java.util.function.Supplier functional interface as a Spring Bean. For example, the following is a publisher that publishes messages as a String:

```
@Bean
public Supplier<String> myPublisher() {
    return () -> new Date().toString();
}
```

Introduction:Microservices:Spring Cloud Stream

- **Spring Cloud Stream:Receiving**
  - To implement a publisher, we only need to implement the java.util.function.Supplier functional interface as a Spring Bean.

```
@Bean
public Consumer<String> mySubscriber() {
    return s -> System.out.println("ML RECEIVED: " + s);
}
```

Introduction:Microservices:Spring Cloud Stream

- **Spring Cloud Stream:Processing**
  - It is also possible to define a Spring Bean that processes messages, meaning that it both consumes and publishes messages

```
@Bean
public Function<String, String> myProcessor() {
    return s -> "ML PROCESSED: " + s;
}
```

Introduction:Microservices:Spring Cloud Stream

- **Spring Cloud Stream:Config**
  - To make Spring Cloud Stream aware of these functions we need to declare them using the spring.cloud.function.definition configuration property

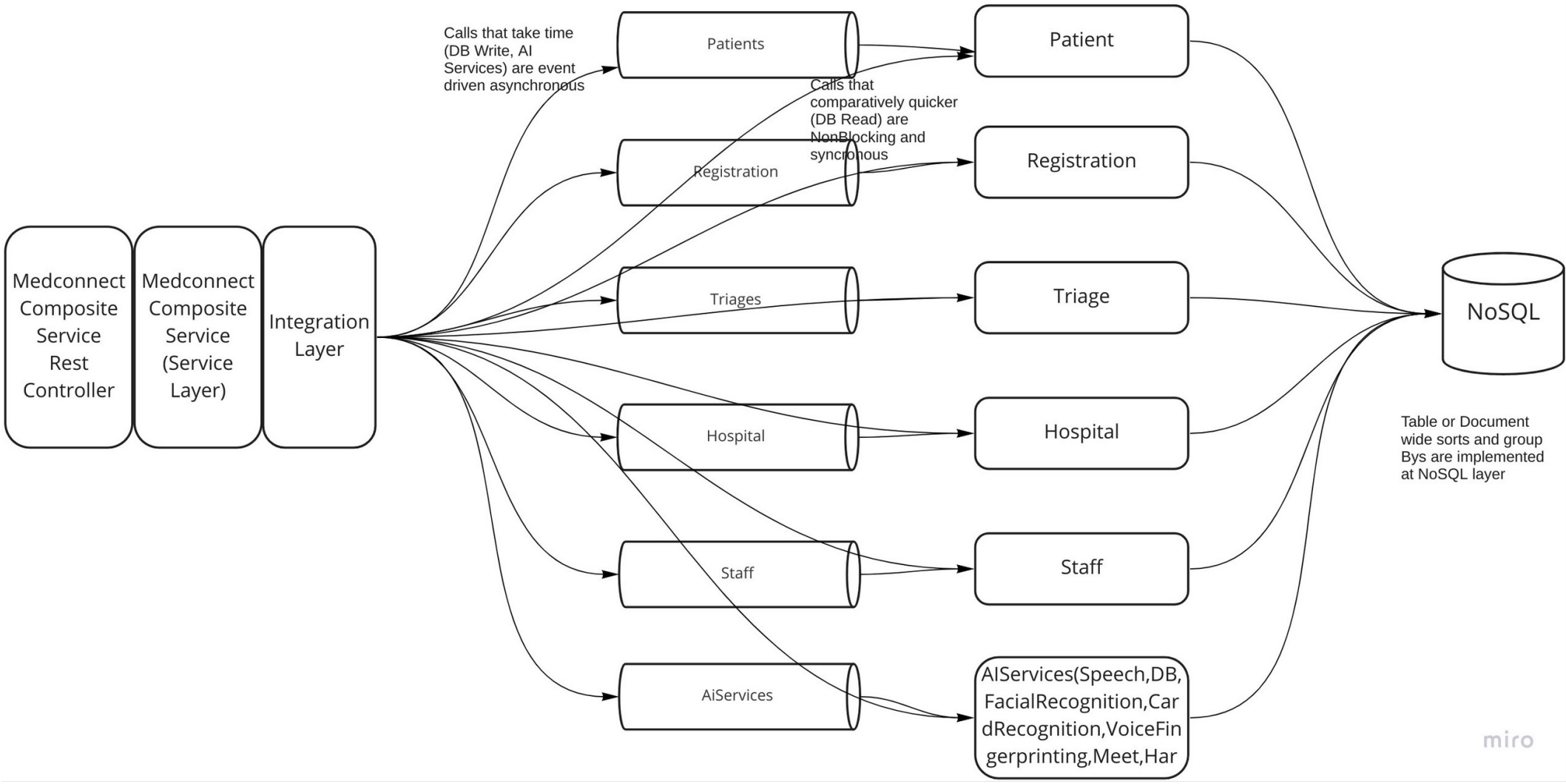
```
spring.cloud.function:
  definition: myPublisher;myProcessor;mySubscriber
```

Introduction:Microservices:Spring Cloud Stream

- **Spring Cloud Stream:Config**
  - To connect our three functions so that our processor consumes messages from our publisher and our subscriber consumes messages from the processor, we can supply the following configuration

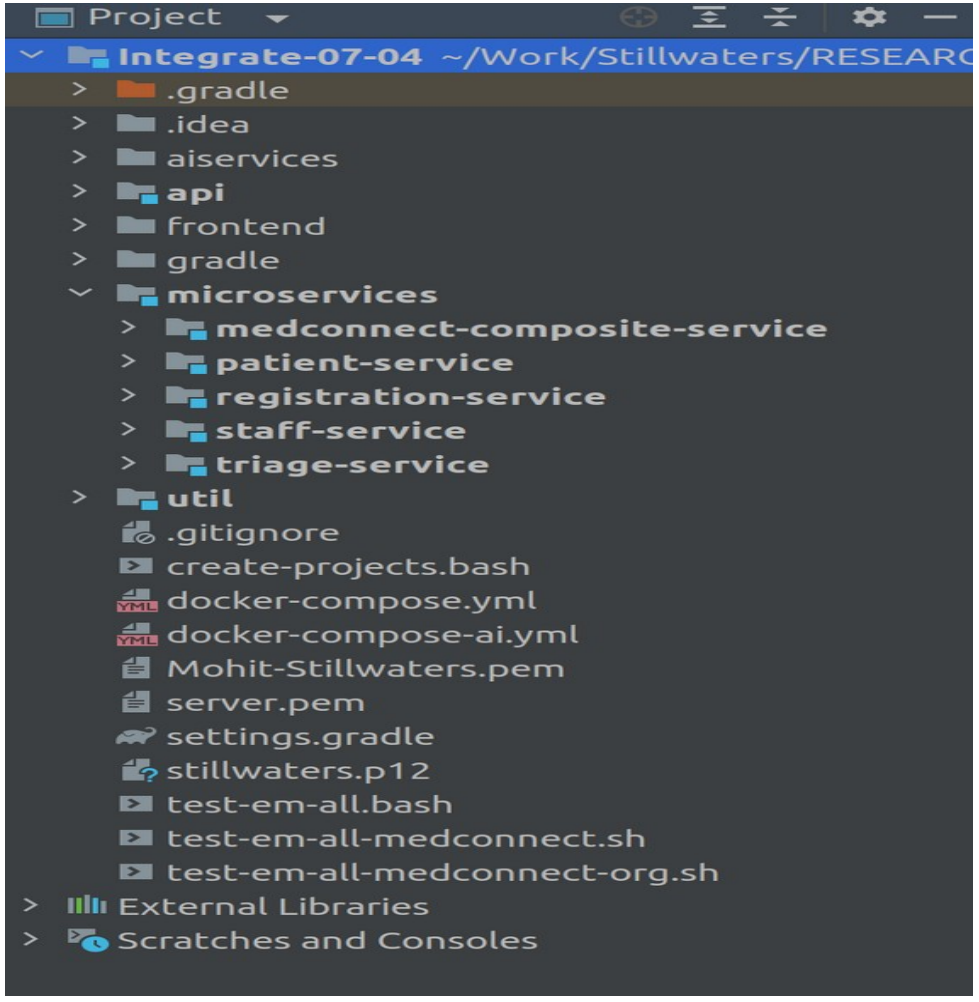
```
spring.cloud.stream.bindings:
  myPublisher-out-0:
    destination: myProcessor-in
  myProcessor-in-0:
    destination: myProcessor-in
  myProcessor-out-0:
    destination: myProcessor-out
  mySubscriber-in-0:
    destination: myProcessor-out
```

Introduction:Microservices:Architecture



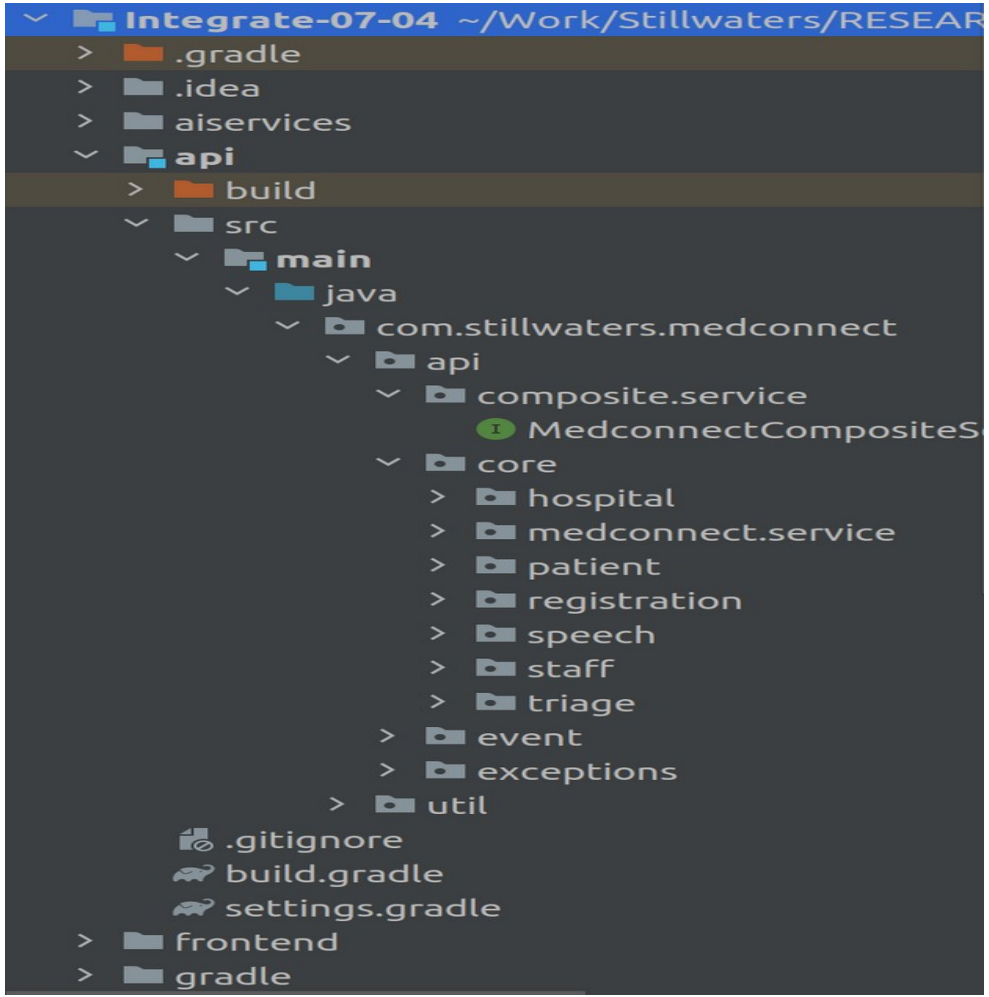
Introduction:Microservices:Architecture

- **Project Structure**



Introduction:Microservices:Architecture

- **Project Structure**



Introduction:Microservices:Architecture

- **Project Structure:**
  - API
  - Util
  - PatientService
  - MedconnectComposite
    - Properties
    - Integration component
    - Composite API implementation
    - The global REST controller exception handler
    - Error handling in API implementations
    - Error handling in the API client
  - Adding automated microservice tests in isolation

Introduction:Microservices:Architecture

- **Project Structure:**
  - OpenAPI
  - Actuator